Чёрная магия JIT-компиляции

Алексей Рагозин
alexey.ragozin@gmail.com





"Any sufficiently advanced technology is indistinguishable from magic"

Arthur C. Clarke



В докладе

- Скорость динамических языков
  – всё не так просто, как кажется
- Грязные трюки JIT-компиляции
- Интерпретаторы
  – а так ли нужна компиляция?
- Graal / Truffle – технологии будущего?



Три кита ООП

- Инкапсуляция
- Полиморфизм
- Наследование



Старый добрый C++

Виртуальный вызов метода

Переменная

Object reference

Объект
в памяти

VTABLE

Таблица
виртуальных методов

00: methodA
01: methodB
02: methodC
03: methodD
04: methodE

Сегмент кода



Три кита ООП

- Инкапсуляция
- Полиморфизм
- ~~Наследование~~



Статические языки XXI века

Полиморфный вызов метода (Rust, GO)

Переменная

VTABLE reference
Object reference

Объект
в памяти

Таблица
виртуальных методов

00: methodA
01: methodB
02: methodC
03: methodD
04: methodE

Сегмент кода

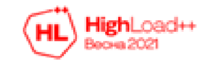


Цена непредсказуемости

Branch misprediction penalty

- Intel Core2 – **15 cycles**
- Intel Nehalem – **17 cycles**
- Intel Sandy/Ivy bridge – **15 cycles**
- Intel Haskwell / Broadwell / Skylake – **15 - 20 cycles**
- AMD K8 / K10 – **13 cycles**
- AMD Buldozer – **19 - 22 cycles**
- AMD Ryzen – **18 cycles**

http://www.agner.org/optimize/microarchitecture.pdf



Цена непредсказуемости

Memory access timings (Skylake)

- L1 cache – 4 cycle
- L2 cache – 14 cycle
- L3 cache – 34-85 cycles
- RAM – 50-100 ns



Непредсказуемость поведения кода
— основной вызов JIT-компилятору



Как реализовать свой язык?

Интерпретатор

Исходный код

The source code for a simple computer program written in the C programming language. When compiled and run, it will give the output "Hello, world!".
A programming language is a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.

AST

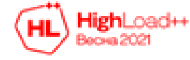
Байт-код

```
 0 getstatic com.oracle.
 3 iconst_0
 4 anewarray java.lang.O
 7 invokevirtual java.la
10 checkcast java.net.So
13 areturn
14 astore_0
15 new java.lang.Asserti
18 dup
19 aload_0
20 invokespecial java.la
23 athrow
24 astore_0
25 new java.lang.Asserti
28 dup
29 aload_0
30 invokespecial java.la
```

Интерпретатор байт-кода

Машинный код

```
0111010010
1110100101
1101001011
1010010111
0100101110
1001011101
001011
010111
```



JIT компиляция

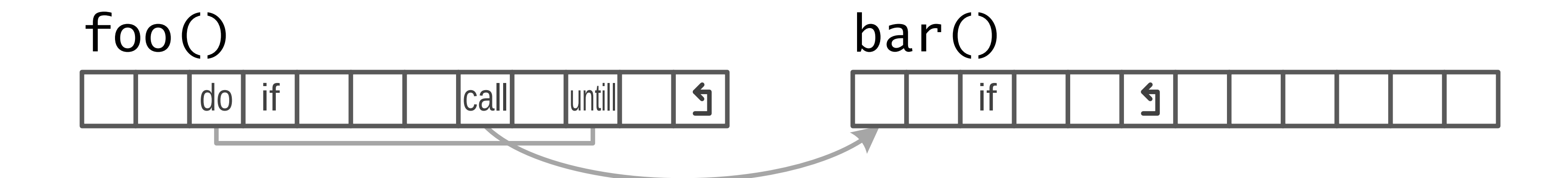
Классический подход

– компиляция методов/функций

Трассирующая компиляция

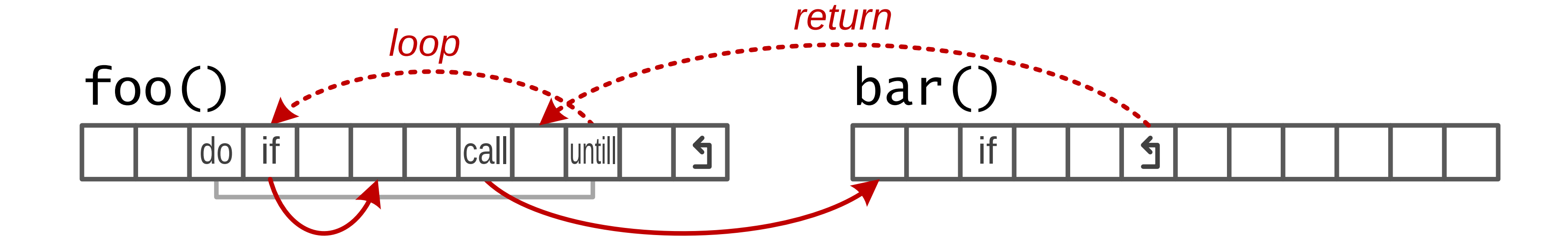


Трассирующий компилятор



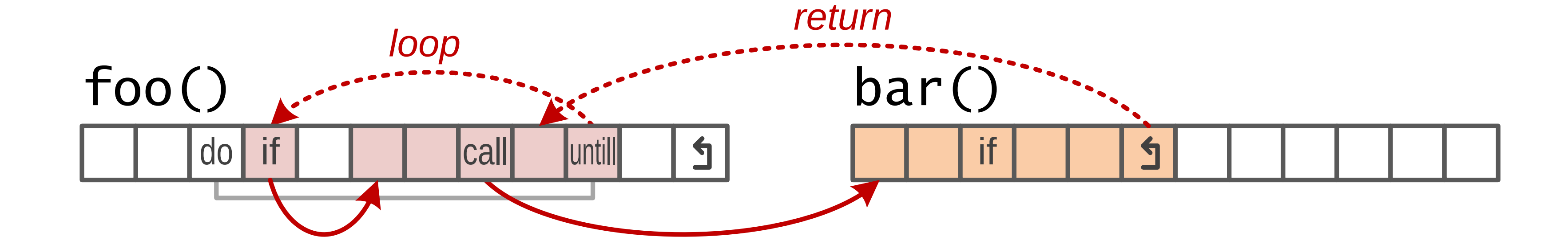


Трассирующий компилятор



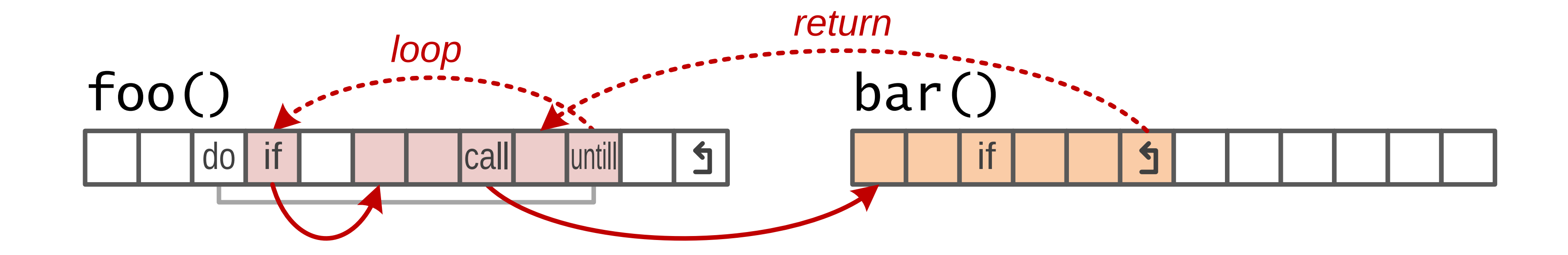


Трассирующий компилятор

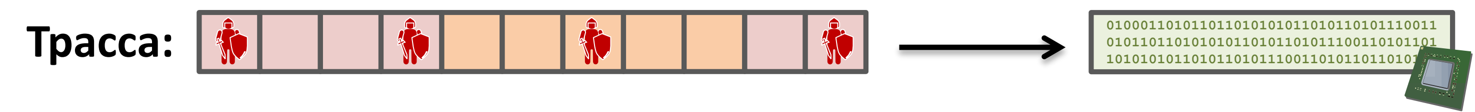




Трассирующий компилятор



Линейная последовательность операций



Нарушение гарда переводит выполнение в режим интерпретации

Компилируется в блок машинного кода



Инлайн-кэш вызовов

```
int size = list.size();
```



Мономорфная точка вызова

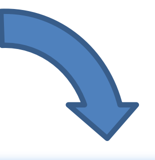
Псевдокод

```
int size;
if (list.class == ArrayList.class) {
 size = ArrayList::size(list);
}
else {
 size = VM::dispatch(List::size, list);
}
```



Инлайн-кэш доступа к свойству

```
obj.value = x
```



LuaJIT-трасса

```
HREFK:  if (hash[17].key != key) goto exit
HSTORE: hash[17].value = x
```

Честное выполнение

- Вычисление хэша
- Деление по модулю
- Сравнение ключа по индексу

Синергия со спекулятивным и суперскалярным выполнением на современном ЦПУ



Трассирующие JIT на практике

Платформы, использующие трассирующий JIT

- ~~Flash~~
- ~~Mozilla TraceMonkey~~
- PyPy / RPython
- LuaJIT

Проблемы трассирующего JIT'a

- Трассировка тормозит интерпретатор
- Очень медленный “разогрев”



Особенности JIT-компилятора

“Многослойность”

- Интерпретатор + Быстрый компилятор + Оптимизирующий компилятор

Инкрементальная компиляция

- Код компилируется по мере выполнения
- Оптимизируется только горячий код

Оптимизация на основании поведения кода

- Сбор информации о типах в точках вызовов
- Спекулятивная специализация кода в оптимизирующем компиляторе

Динамическая деопитимизация

- Если спекуляции оказались неверны



Грязные трюки JIT

Спекулятивное выполнение

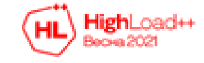
- Инлайн-кэширование динамических вызовов
- Инлайн-кэширование структурной информации
- Спекулятивное выполнение ветвлений

Агрессивный инлайнинг кода

- Возможен за счёт спекуляции

Оптимизация (устранение) аллокаций в куче

- Более эффективен за счёт инлайнинга

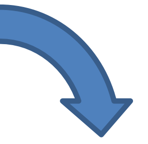


Scalar replacement

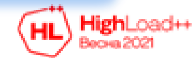
```
double length() {
  return distance(
  new Point(this.ax, this.ay),
  new Point(this.bx, this.by));
}

double distance(Point a, Point b) {
  double w = a.x - b.x;
  double h = a.y - b.y;
  return Math.sqrt(w*w + h*h);
}
```

- ✓ включение distance
- ✓ объекты a и b не убегают
- ✓ объекты a и b декомпозируются в набор полей



```
double length() {
  double w = this.ax - this.bx;
  double h = this.ay - this.by;
  return Math.sqrt(w*w + h*h);
}
```



А нужна ли магия?

Что, если генерировать код по-простому?

Интерпретатор может оказаться быстрее!

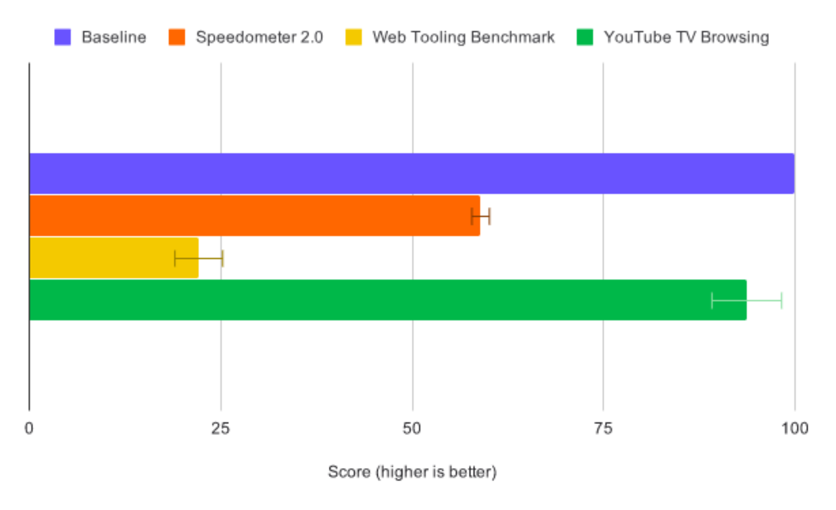
- V8 TurboFan Vs. V8 Ignition
- HIPHOPVM Vs. PHP 7
- kdb
- “Building fast interpreters in Rust” @blog.cloudflare.com

https://blog.cloudflare.com/building-fast-interpreters-in-rust/



V8 TurboFan Vs. Ignition

Since JIT-less mode disables the optimizing compiler, it comes with a performance penalty. We looked at a variety of benchmarks to better understand how V8's performance characteristics change. Speedometer 2.0 is intended to represent a typical web application; the Web Tooling Benchmark includes a set of common JS developer tools; and we also include a benchmark that simulates a browsing workflow on the Living Room YouTube app. All measurements were made locally on an x64 Linux desktop over 5 runs.



JIT-less vs. default V8. Scores are normalized to 100 for V8's default configuration.

https://v8.dev/blog/jitless



HIPHOPVM Vs. PHP7

PHP 7

Statistics - QuickStorm of [redacted]

Total Requests	2,500
Total Errors	0 (0%)
Peak RPS	7
Average RPS	4.17
Peak Response Time(ms)	207
Average Response Time(ms)	81
Total Data Transferred(MB)	46.54
Peak Throughput(kB/s)	130.31
Average Throughput(kB/s)	77.57

HHVM

Statistics - QuickStorm of [redacted]

Total Requests	2,600
Total Errors	0 (0%)
Peak RPS	7
Average RPS	4.33
Peak Response Time(ms)	14,222
Average Response Time(ms)	62
Total Data Transferred(MB)	48.86
Peak Throughput(kB/s)	131.56
Average Throughput(kB/s)	81.44

https://www.wpoven.com/blog/hhvm-vs-php-7-performance-showdown-wordpress-nginx



Интерпретатор HotSpot JVM

- Для каждой инструкции байт-кода написана процедура на ассемблере
- Байт-код – индекс в таблице адресов процедур
- Вход в процедуру инструкции - jump
- Каждая процедура заканчивается jump на вход интерпретатора



Всё так сложно ☹

JIT-компилятор

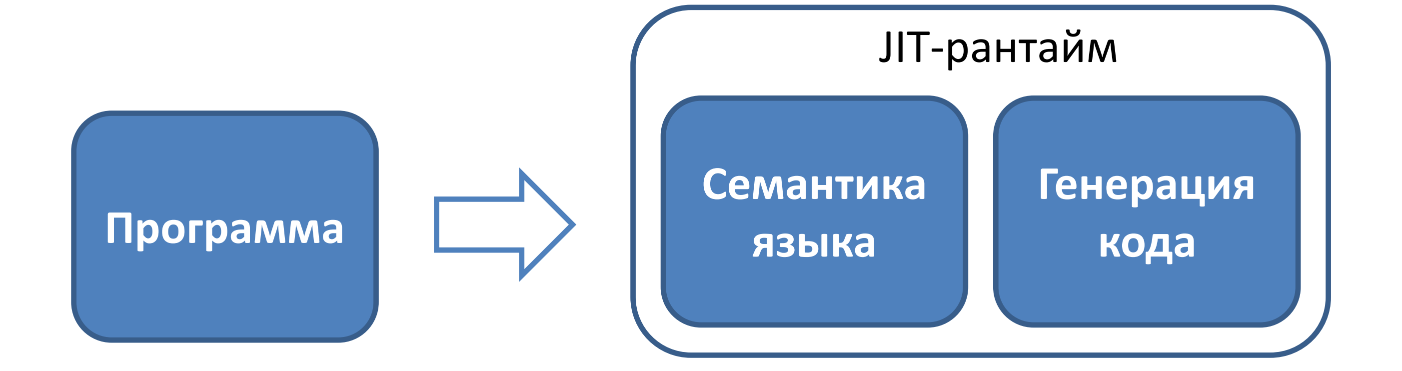
- Генерация кода
- Специфика языка
- Оптимизации

Интерпретатор

- Супер-оптимизированный код
- “Заточка” под архитектуру ЦПУ

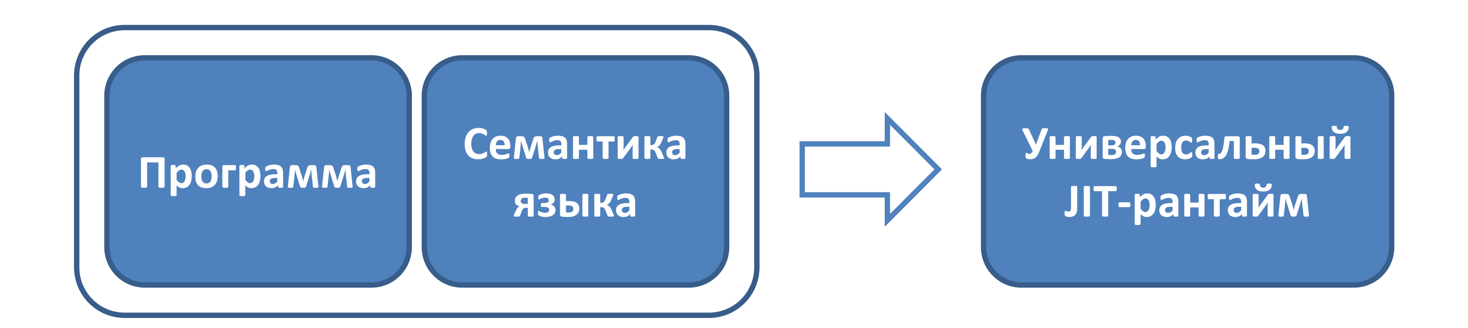


А что если?


Программа
JIT-рантайм
Семантика языка
Генерация кода

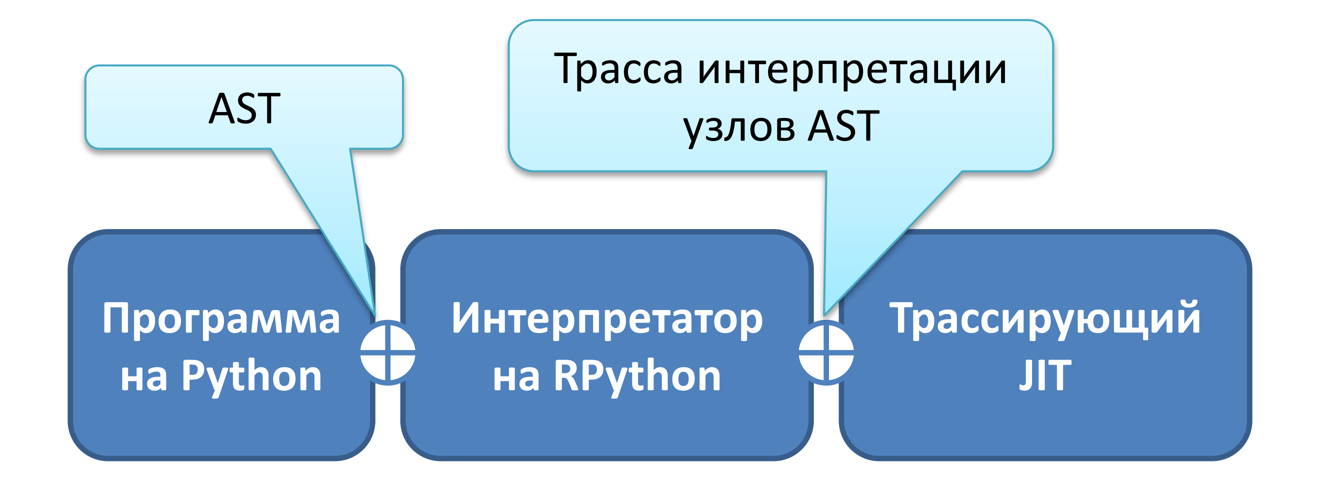

HL
HighLoad++
Весна 2021

А что если?


Программа
Семантика языка
Универсальный JIT-рантайм

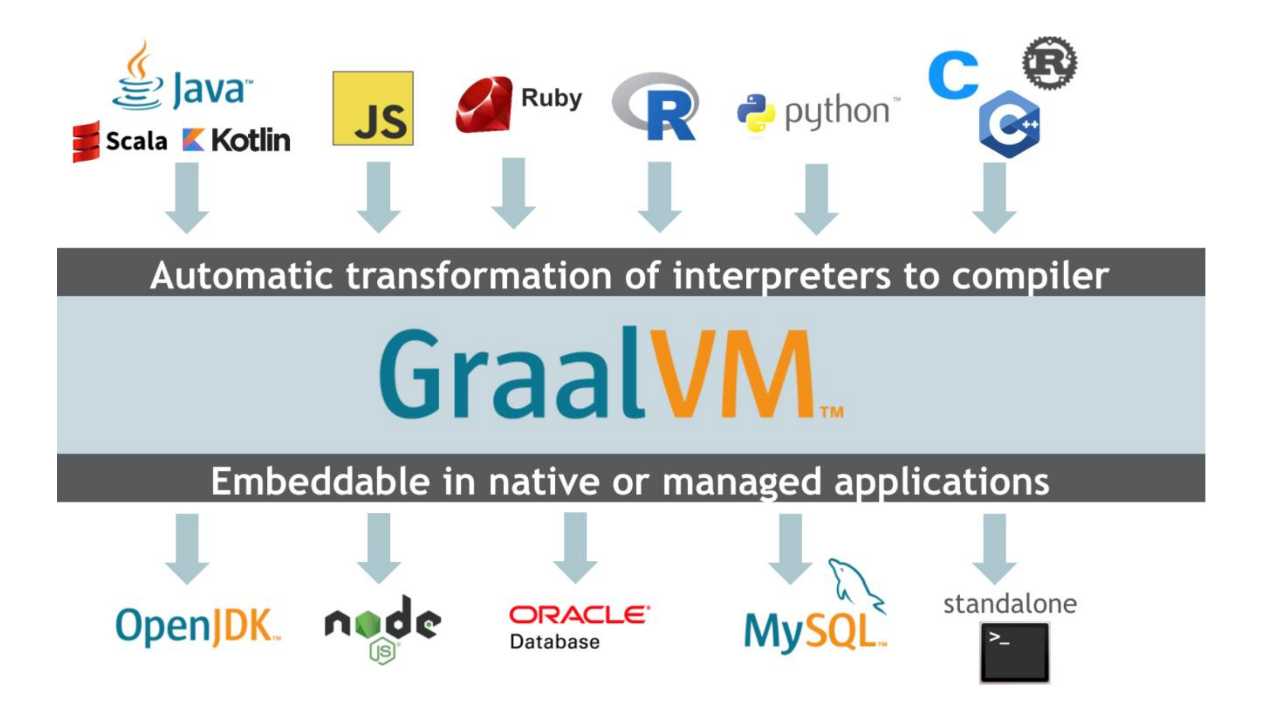

HL
HighLoad++
Весна 2021

PyPy – Питон на Питоне





Graal & Truffle

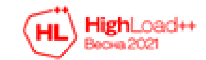


https://www.graalvm.org

HighLoad++

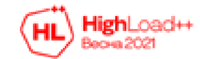
Graal & Truffle

- Оптимизирующий JIT-компилятор
- Написанный на Java
- 150+ видов/фаз оптимизации
- Понимает JVM-байт-код, но не только
- Использует промежуточное графовое представление код
- Может использоваться вместо C2 в JDK 11
- Truffle – языковой фасад для Graal



Graal + Truffle





TruffleRuby

iterations per minute

0
5
10
15
20
25
30
35
40

2.0.0p648
2.2.3p173
rbx-2.5.8
JRuby 9.0.3.0
JRuby 9.0.3.0 + id
truffleruby old
2.4.0p0
rbx-3.69
JRuby 9.1.7.0
JRuby 9.1.7.0 + id
truffleruby

https://pragtob.wordpress.com/2017/01/24/benchmarking-a-go-ai-in-ruby-cruby-vs-rubinius-vs-jruby-vs-truffle-a-year-later/



Truffle

Реализуем трюфельный JIT-рантайм

1. Пишем интерпретатор AST на Java
2. Включаем Graal-компилятор в JVM
3. Добавляем JIT-оптимизации декларативно



Truffle

```
public final ConditionProfile condition = ConditionProfile.createCountingProfile();
...
@Override
public void executeVoid(VirtualFrame frame) {

  if (condition.profile(evaluateCondition(frame))) {
    /* Execute the then-branch. */
    thenPartNode.executeVoid(frame);
  } else {
    /* Execute the */
    if (elsePartNode != null) {
      elsePartNode.executeVoid(frame);
    }
  }
}
```

https://github.com/graalvm/simplelanguage



Truffle

```
@Specialization(limit = "INLINE_CACHE_SIZE", //
                guards = "function.getCallTarget() == cachedTarget", //
                assumptions = "callTargetStable")
@SuppressWarnings("unused")
protected static Object doDirect(SLFunction function, Object[] arguments,
     @Cached("function.getCallTargetStable()") Assumption callTargetStable,
     @Cached("function.getCallTarget()") RootCallTarget cachedTarget,
     @Cached("create(cachedTarget)") DirectCallNode callNode) {
  /* Inline cache hit, we are safe to execute the cached call target. */
  return callNode.call(arguments);
}

@Specialization(replaces = "doDirect")
protected static Object doIndirect(SLFunction function, Object[] arguments,
     @Cached("create()") IndirectCallNode callNode) {
 /* SL has a quite simple call lookup: just ask the function for
  * the current call target, and call it. */
  return callNode.call(function.getCallTarget(), arguments);
}
```

https://github.com/graalvm/simplelanguage

HL HighLoad++ Весна 2021

Ссылки

LuaJIT

https://web.archive.org/web/20180721041742/http://article.gmane.org/gmane.comp.lang.lua.general/58908

https://github.com/lukego/blog/issues?q=is%3Aissue+is%3Aopen+label%3Aluajit

Incremental Dynamic Code Generation with Trace Trees

https://www.cs.montana.edu/ross/classes/fall2009/cs550/resources/Tracemonkey-01.pdf

V8 Design blog

https://v8.dev/blog | https://v8.dev/blog/jitless | https://mrale.ph/

RPython

https://rpython.readthedocs.io/en/latest/jit/index.html

http://tratt.net/laurie/research/pubs/papers/bolz_tratt__the_impact_of_metatracing_on_vm_design_and_implementation.pdf

Интерпретаторы

https://blog.cloudflare.com/building-fast-interpreters-in-rust/

https://badootech.badoo.com/when-pigs-fly-optimising-bytecode-interpreters-f64fb6bfa20f



Ссылки

Graal

https://github.com/oracle/graal/blob/master/docs/Publications.md

https://www.graalvm.org

https://chrisseaton.com/rubytruffle/pppj14-om/pppj14-om.pdf – object layout for TruffleRuby

https://www.youtube.com/watch?v=FJY96_6Y3a4 – 3 hours Truffle introduction

https://github.com/graalvm/graal-js-jdk11-maven-demo – out-of box Java 11 + Graal JIT setup

Partial escape analysis

http://www.ssw.uni-linz.ac.at/Research/Papers/Stadler14/Stadler2014-CGO-PEA.pdf



Спасибо!

https://blog.ragozin.info
https://aragozin.timepad.ru

Алексей Рагозин
alexey.ragozin@gmail.com

